CHAPTER 1 Introduction

If I have seen further it is by standing upon the shoulders of Giants
– Sir Isaac Newton

JAVA is a general-purpose, concurrent class-based object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. Java is based on C and parts of C++, with many omissions and a few additional ideas from other languages. Java is intended to be a production language, not a research language, and so, as C. A. R. Hoare suggested in his classic paper on language design, Java avoids including new and untested features.

Java is strongly typed. This specification clearly distinguishes between errors that can and must be detected at compile time, and those that occur at run time. Compile time normally consists of translating Java programs into a machine-independent byte-code representation. Run-time activities include loading and linking of the classes needed to execute a program, optional machine code generation and dynamic optimization of the program, and actual program execution.

Java is a relatively high-level language, in that details of the machine representation are not available through the language. It includes automatic storage management, typically using a garbage collector, to avoid the safety problems of explicit deallocation (as in C's free or C++'s delete). High-performance garbage-collected implementations of Java can have bounded pauses to support systems programming and real-time applications. Java does not include any unsafe constructs, such as array accesses without bounds checking. Unsafe constructs would cause a program to behave in a way not deducible from the language specification.

Java is normally compiled to a bytecoded instruction set and binary format defined in *The Java Virtual Machine* (Addison-Wesley, 1996). This instruction set may be directly executed by an interpreter, or, for faster execution, higher performance and even highly optimized machine code can be generated at execution time. Most implementations of Java for general-purpose programming will support the additional packages defined in the series of books under the general title *The Java Application Programming Interface* (Addison-Wesley).
This Java Language Specification is organized as follows:

Chapter 2 describes grammars and the notation used to present the lexical and syntactic grammars for Java.

Chapter 3 describes the lexical structure of Java, which is based on C and C++. Java is written in the Unicode character set. Java supports the writing of Unicode characters on systems that support only ASCII.

Chapter 4 describes Java's types, values, and variables. Java's types are the primitive types and reference types.

The primitive types are defined to be the same on all machines and in all implementations, and are various sizes of two's-complement integers, single- and double-precision IEEE 754 standard floating-point numbers, a boolean type, and a Unicode character char type. Values of the primitive types do not share state.

Java's reference types are the class types, the interface types, and the array types. The reference types are implemented by dynamically created objects that are either instances of classes or arrays. Many references to each object can exist. All objects (including arrays) support the methods of the standard class Object, which is the (single) root of the class hierarchy. A predefined String class supports Unicode character strings. Standard classes exist for wrapping primitive values inside of objects.

Variables are typed storage locations. A variable of a primitive type holds a value of that exact primitive type. A variable of a class type can hold a null reference or a reference to an object whose type is any subclass of that class type. A variable of an interface type can hold a null reference or a reference to an instance of any class that implements the interface. A variable of an array type can hold a null reference or a reference to an array. A variable of class type Object can hold a null reference or a reference to any object, whether class instance or array.

Chapter 5 describes Java's conversions and numeric promotions. Conversions change the compile-time type and, sometimes, the value of an expression. Numeric promotions are used to convert the operands of a numeric operator to a common type where an operation can be performed. There are no loopholes in the language; casts on reference types are checked at run time to ensure type safety.

Chapter 6 describes declarations and names, and how to determine what names mean (denote). Java does not require types or their members to be declared before they are used. Declaration order is significant only for local variables and the order of initializers of fields in a class or interface.

Java provides control over the scope of names and supports limitations on external access to members of both packages and classes. This helps when writing large programs by distinguishing the implementation of a type from its users and those who extend it. Standard naming conventions that make for more readable programs are described here.

Chapter 7 describes the structure of a Java program, which is organized into packages similar to the modules of Modula. The members of a package are compilation units and subpackages. Compilation units contain type declarations and can import types from other packages to give them short names. Packages have names in a hierarchical name space, and can use the Internet domain name system to form unique names.

Chapter 8 describes Java's classes. The members of classes are fields (variables) and methods. Class variables exist once per class. Class methods operate without reference to a specific object. Instance variables are dynamically created in objects that are instances of classes. Instance methods are invoked on instances of classes, which instances become the current object this during their execution, supporting the object-oriented programming style.

Classes support single implementation inheritance, in which the implementation of each class is derived from that of a single superclass, and ultimately from the class Object. Variables of a class type can reference an instance of any subclass of a class, allowing new types to be used with existing methods, polymorphically.

Classes support concurrent programming with synchronized methods. Methods declare the checked exceptions that can arise from their execution, providing compile-time checking that ensures exceptional conditions are handled. Objects can declare a finalize method that will be invoked before the objects are discarded by the garbage collector, allowing the objects to clean up their state.

For simplicity, Java has neither declaration "headers" separate from the implementation of a class nor separate type and class hierarchies.

Although Java does not include parameterized classes, the semantics of arrays is that of a parameterized class with some syntactic sugar. Like the programming language Beta, Java uses a run-time type check when storing references in arrays to ensure complete type safety.

Chapter 9 describes Java's interface types, which declare a set of abstract methods and constants. Classes that are otherwise unrelated can implement the same interface type. A variable of an interface type can contain a reference to any object that implements the interface. Multiple interface inheritance is supported.

Chapter 10 describes Java arrays. Array accesses include bounds checking. Arrays are dynamically created objects and may be assigned to variables of type Object. Java supports arrays of arrays, rather than multidimensional arrays.

Chapter 11 describes Java's exceptions, which are non-resuming and fully integrated with the language semantics and concurrency mechanisms. There are three kinds of exceptions: checked exceptions, run-time exceptions, and errors. The compiler ensures that checked exceptions are properly handled by requiring that a method or constructor can result in a checked exception only if it declares it. This provides compile-time checking that exception handlers exist, and aids programming in the large. Most user-defined exceptions should be checked exceptions. Run-time exceptions result from bugs in the program detected by the Java Virtual Machine, such as NullPointerException. Errors result from failures detected by the virtual machine for example LinkageError or OutOfMemoryError. Most simple programs do not try to handle errors.

Chapter 12 describes activites that occur during execution of a Java program. A Java program is stored as binary files representing compiled classes and interfaces. These binary files can be loaded into a Java Virtual Machine, linked to other classes and interfaces, and initialized.

After initialization, class methods of the class may be invoked. Some classes may be instantiated to create new objects of the class type. Objects that are class instances also contain an instance of each superclass of the class, and the creation of such an object involves recursive creation of these superclass instances.

When an object is no longer referenced, it may be reclaimed by the garbage collector. If an object declares a finalizer, the finalizer is executed before the object is reclaimed to give the object a last chance to clean up resources that would not otherwise be released. When a class is no longer needed it may be unloaded; if a class finalizer is declared it is given a chance to clean up first. Objects and classes may be finalized on exit of the Java Virtual Machine.

This chapter includes a specification of the impact of changes in types on other types that use the changed types but are not recompiled. These considerations are of interest to developers of types that are to be widely distributed, in a continuing series of versions, into the Internet. Good program development environments will automatically recompile dependent whenever a type is changed, so most programmers will not need to be concerned about these details.

Chapter 13 describes Java's blocks and statements. Java has no goto, but includes labeled break and continue statements. Unlike C, Java requires boolean expressions in control-flow statements, and does not convert types to boolean implicitly, in the hopes of catching more errors at compile time. A synchronized statement provides basic object-level monitor locking. A try statement can include catch and finally clauses to protect against non-local control transfers.

Chapter 14 describes Java's expressions. Java fully specifies the (apparent) order of evaluation of expressions, for increased determinism and portability. Overloaded methods and constructors are resolved at compile-time by picking the most specific method or constructor from those which are applicable. Java chooses which method or constructor by using the same basic algorithm used in languages with richer dispatching, such as CLOS and Dylan, for the future.

Chapter 15 describes the precise way in which Java ensures that local variables are definitely set before use. While all other variables are automatically initialized to a default value, Java does not automatically initialize local variables in order to avoid masking bugs.

Chapter 16 describes the semantics of Java threads and locks, which are based on the monitor-based concurrency originally introduced with the Mesa program-

j.12345.doc 5 Wed Mar 27 08:55:12 1996

ming language. Java specifies a memory model for shared-memory multiprocessors that supports high-performance implementations.

Chapter 17 describes the facilities for automatically generating documentation from special comments in Java source code.

Chapter 18 presents a LALR(1) syntactic grammar for Java, and describes the differences between this grammar and the expository grammar used in the body of the language specification that precedes it.

Chapters 19 through 21 are the reference manual for the core of the standard Java application programming interface. These packages must be included in all general purpose Java systems.

Chapter 19 describes the package java.lang. The types defined in java.lang are automatically imported to be available without qualification in all Java programs. They include the primordial Object class; classes such as Integer and Float; which *wrap* the primitive types inside objects; exceptions and errors defined by the language and the Java Virtual Machine; Thread support; metalinguistic classes such as Class and ClassLoader; and the class System that abstracts the host system.

Chapter 20 describes the package java.util, which defines a few basic utility classes, such as a hash table class and a random number generator.

Chapter 21 describes the package java.io, which defines basic input/output facilities, including random access files and streams of values of primitive types.

The book concludes with two indexes: one for the types, methods, and fields defined and described in this specification, and the other a more traditional index.

1.1 Example Programs

Most of the example programs given in the text are ready to be executed by a Java system, and are similar in form to:

```
class Test {
	public static void main(String args[]) {
		for (int i = 0; i < args.length; i++)
			System.out.print(args[i] + " ");
		System.out.println();
	}
}
```

On a Sun workstation, this class, stored in the file Test.java, can be compiled and executed by giving the commands:

```
javac Test.java
java Test Hello, world.
```

producing the output:

Hello, world.

1.2 References

Apple Computer. *Dylan™ Reference Manual.* Apple Computer Inc., Cupertino, California. September 29, 1995. See also http://www.cambridge.apple.com on the World Wide Web.

Bobrow, Daniel G., Linda G. Demichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. *Common Lisp Object System Specification*, X3J13 Document 88-002R, June 1988; appears as Chapter 28 of Steele, Guy. *Common Lisp: The Language*, 2nd ed. Digital Press, 1990, ISBN 1-55558-041-6, 770-864.

Ellis, Margaret A., and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990, reprinted with corrections October 1992, ISBN 0-201-51459-1.

Harbison, Samuel. *Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1992, ISBN 0-13-596396.

Hoare, C.A.R. *Hints on Programming Language Design.* Stanford University Computer Science Department Technical Report No CS-73-403, December 1973. Reprinted in Sigact/Sigplan Symposium on Principles of Programming Languages. Association for Computing Machinery, New York, October 1973.

IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985. Available from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112-5704 USA; 800-854-7179.

Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language,* 2nd ed. Prentice Hall, Englewood Cliffs, New Jersey, 1988, ISBN 0-13-110362-8.

Madsen, Ole Lehrmann, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, Massachusetts, 1993, ISBN 0-201-52430.

Mitchell, James G, William Maybury, and Richard Sweet. *The Mesa Programming Language*, Version 5.0. Xerox PARC, Palo Alto, California, CSL 79-3, April 1979.

Stroustrup, Bjarne. *The C++ Progamming Language,* 2nd ed. Addison-Wesley, Reading, Massachusetts, 1991, reprinted with corrections, January 1994, ISBN 0-201-53992-6.

CHAPTER 2
Grammars

Grammar, which knows how to control even kings...
– Molière, Les Femmes Savantes, 1672
Act II, sc. vi.

THIS specification uses context-free grammars to define the lexical and syntactic structure of a Java program. This chapter describes such grammars and the notation used in presenting them.

2.1 Context-free Grammars

A context-free *grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the infinite set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

2.2 The Lexical Grammar

A *lexical grammar* for Java is given in §3. This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions starting from the goal symbol *Input* (§3.5) that describe how sequences of Unicode characters (§3.1) are translated into a sequence of input elements (§3.5).

These input elements, with white space (§3.6) and comments (§3.7) discarded, form the terminal symbols for the syntactic grammar for Java and are called the Java tokens (§3.5). These tokens are the keywords (§3.8), identifiers (§3.9), literals (§3.10), separators (§3.11), and operators (§3.12) of the Java language.

2.3 The Syntactic Grammar

The *syntactic grammar* for Java is given in chapters 4, 6–10, 13 and 14. This grammar describes a set of productions starting from the goal symbol *CompilationUnit* (§7.3) that describes how sequences of tokens can form syntactically correct Java programs.

A LALR(1) version of the syntactic grammar is presented in chapter 18. We have written the grammar in the body of the document to be both close to the LALR(1) grammar and readable.

2.4 Grammar Notation

Terminal symbols are shown in `fixed width` font in the productions of the lexical and syntactic grammars, and throughout this specification whenever the text is directly referring to such a terminal symbol. These are to appear in a program exactly as written.

Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the syntactic definition:

IfThenStatement:
    `if (` *Expression* `)` *Statement*

states that the nonterminal *IfThenStatement* represents the token `if` followed by a left parenthesis token followed by an *Expression* followed by a right parenthesis token followed by a *Statement*. As another example, the syntactic definition:

ArgumentList:
    Argument
    ArgumentList `,` *Argument*

states that an *ArgumentList* may represent either a single *Argument* or an *ArgumentList* followed by a comma followed by an *Argument*. This definition of *ArgumentList* is *recursive*, that is to say, it is defined in terms of itself. The result is that an *ArgumentList* may contain any positive number of arguments. Such recursive definitions of nonterminals are common.

The subscripted suffix $_{opt}$, which may appear after a terminal or nonterminal, indicates an *optional symbol*. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it. This means that:

BreakStatement:
  break *Identifier*$_{opt}$;

is a convenient abbreviation for:

BreakStatement:
  break ;
  break *Identifier* ;

and that:

ForStatement:
  for (*ForInit*$_{opt}$; *Expression*$_{opt}$; *ForUpdate*$_{opt}$) *Statement*

is a convenient abbreviation for:

ForStatement:
  for (*ForInit* ; *Expression*$_{opt}$; *ForUpdate*$_{opt}$) *Statement*
  for (; *Expression*$_{opt}$; *ForUpdate*$_{opt}$) *Statement*

which in turn is an abbreviation for:

ForStatement:
  for (*ForInit* ; *Expression* ; *ForUpdate*$_{opt}$) *Statement*
  for (*ForInit* ; ; *ForUpdate*$_{opt}$) *Statement*
  for (; *Expression* ; *ForUpdate*$_{opt}$) *Statement*
  for (; ; *ForUpdate*$_{opt}$) *Statement*

which in turn is an abbreviation for:

ForStatement:
  for (*ForInit* ; *Expression* ; *ForUpdate*) *Statement*
  for (*ForInit* ; *Expression* ;) *Statement*
  for (*ForInit* ; ; *ForUpdate*) *Statement*
  for (*ForInit* ; ;) *Statement*
  for (; *Expression* ; *ForUpdate*) *Statement*
  for (; *Expression* ;) *Statement*
  for (; ; *ForUpdate*) *Statement*
  for (; ;) *Statement*

so the nonterminal *ForStatement* actually has eight alternative right-hand sides.

When the words "one of" follow the colon in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition. For example, the lexical grammar for Java contains the production:

ZeroToThree: one of
  0 1 2 3

which is merely a convenient abbreviation for:

ZeroToThree:
  0
  1
  2
  3

As an extension of this notation, when such a "one of" alternative in a lexical production appears to be a token, it represents the sequence of characters that would make up such a token. For example, the definition:

Keyword: one of
  if else for while

in a lexical grammar production is shorthand for:

Keyword:
  i f
  e l s e
  f o r
  w h i l e

The right-hand side of a lexical production may indicate that certain expansions are not permitted by using the phrase "but not" and then naming the excluded expansions, as in the productions for *InputCharacter* (§3.4), *NotStar* (§3.7), and *Identifier* (§3.9):

InputCharacter:
  UnicodeInputCharacter, but not CR and not LF

Identifier:
  IdentifierName, but not a *Keyword* or *BooleanLiteral*

Finally, a few nonterminal symbols are described by a descriptive phrase in "roman type" in cases where it would be unwieldly or impractical to list all the alternatives:

RawInputCharacter:
  any Unicode character

CHAPTER 3 Lexical Structure

Lexicographer — A writer of dictionaries, a harmless drudge.
– Samuel Johnson, Dictionary, 1755

THIS chapter describes the lexical structure of Java by specifying the language's lexical grammar (§2.2).

3.1 Unicode

Java programs are written using the *Unicode* character encoding, version 1.1, as specified in *The Unicode Standard: Worldwide Character Encoding*, Version 1.0, Volume 1, ISBN 0-201-56788-1, and Volume 2, ISBN 0-201-60845-6, and the update information about Unicode 1.1 available at ftp://unicode.org. (There are a few minor errors in this update information; see §19.4 for corrections.)

Except for comments (§3.7) and identifiers (§3.9) and the contents of character and string literals (§3.10.4, §3.10.5), all input elements (§3.5) in a Java program are formed from only *ASCII* characters. ASCII (ANSI X3.4) is the American Standard Code for Information Interchange. The first 128 characters of the Unicode character encoding are the ASCII characters.

Java defines a standard Unicode escape sequence (§3.3) that can be used to represent any Unicode character using only ASCII characters.

3.2 Lexical Translations

This chapter describes the translation of a raw Unicode character stream into a sequence of Java tokens (§3.5), using the following three lexical translations, which are applied in turn:

1. A translation of Unicode escapes (§3.3) in the raw stream of Unicode characters to the corresponding Unicode character. Unicode escapes have the form

\u*xxxx*, where *xxxx* is a hexadecimal value, written in ASCII characters. This translation allows any Java program to be input in ASCII.

2. A translation of the Unicode stream resulting from step 1 into a stream of input characters and line terminators (§3.4).
3. A translation of the stream of input characters and line terminators resulting from step 2 into a sequence of Java input elements (§3.5) which, after discarding white space (§3.6) and comments (§3.7), comprise the tokens (§3.5) that are the terminal symbols of the syntactic grammar (§2.3) for Java.

In these lexical translations Java chooses the longest possible translation at each step, even if the result does not ultimately make a correct Java program, while another lexical translation would. Thus the input characters `a--b` are tokenized (§3.5) as `a`, `--`, `b`, which is not part of any grammatically correct Java program, even though the tokenization `a`, `-`, `-`, `b` is a part of some grammatically correct Java programs.

3.3 Unicode Escapes

Java implementations first recognize *Unicode escapes* in their input, translating the characters `\u` followed by four hexadecimal digits to the Unicode character with the indicated hexadecimal value, and passing all other characters unchanged. This translation step results in a sequence of Unicode input characters:

UnicodeInputCharacter:
  UnicodeEscape
  RawInputCharacter

UnicodeEscape:
  \ *UnicodeMarker HexDigit HexDigit HexDigit HexDigit*

UnicodeMarker:
  `u`
  UnicodeMarker `u`

RawInputCharacter:
  any Unicode character

HexDigit: one of
  0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

If a `\` is not followed by `u`, then it is treated as a *RawInputCharacter* and remains part of the escaped Unicode stream. If a `\` is followed by `u`, or more than one `u`, and the last u is not followed by four hexadecimal digits, then a compile-time error occurs.

The character produced by a Unicode escape does not participate in further Unicode escape processing. For example, the raw input \u005cu005a results in the six characters \ u 0 0 5 a, because 005c is the Unicode value for \. It does not result in the single character Z, which is Unicode character 005a, because the \ that resulted from the \u005c is not interpreted as the start of a further Unicode escape sequence.

Java specifies a standard way of transforming a Unicode Java program into ASCII that changes a Java program into a form that can be edited by ASCII-based tools. The transformation involves converting any Unicode escapes in the source text of the program to ASCII by adding an extra u—for example, \u*xxxx* becomes \uu*xxxx*—while simultaneously converting non-ASCII characters in the source text to a \u*xxxx* escape containing a single u. The exact Unicode source can later be restored from this ASCII form by converting each escape sequence where multiple u's are present to a sequence of Unicode characters with one fewer u, while simultaneously converting each escape sequence with a single u to the corresponding single Unicode character.

Java systems should use the \u*xxxx* notation as an output format to display Unicode characters when a suitable font is not available.

3.4 Line Terminators

Java implementations next divide the sequence of Unicode input characters into lines by recognizing *line terminators*. This definition of lines determines the line numbers produced by a Java compiler or other Java system component. It also specifies the termination of the // form of a comment (§3.7).

LineTerminator:
  the ASCII LF character, also known as "newline"
  the ASCII CR character, also known as "return"
  the ASCII CR character followed by the ASCII LF character

InputCharacter:
  UnicodeInputCharacter, but not CR and not LF

Lines are terminated by the ASCII characters CR or LF or CR LF. A CR immediately followed by LF is counted as one line terminator, not two. The result is a sequence of line terminators and input characters, which are the terminal symbols for the third step in the tokenization process

3.5 Input Elements and Tokens

Input characters resulting from escape processing (§3.3) and then input line recognition (§3.4) are further reduced to a sequence of *input elements*. Those input elements that are not white space (§3.6) or comments (§3.7) are *tokens*. The tokens are the terminal symbols of the Java syntactic grammar (§2.3).

This process is specified by the following grammar:

Input:
    InputElements$_{opt}$ *Sub*$_{opt}$

InputElements:
    InputElement
    InputElements InputElement

InputElement:
    WhiteSpace
    Comment
    Token

WhiteSpace:
    the ASCII SP character, also known as “space”
    the ASCII HT character, also known as “horizontal tab”
    the ASCII FF character, also known as “form feed”
    LineTerminator

Token:
    Keyword
    Identifier
    Literal
    Separator
    Operator

Sub:
    the ASCII SUB *character, also known as “control-Z”*

White space (§3.6) and comments (§3.7) can serve to separate tokens that, if adjacent, might be tokenized in another manner. For example, the characters `-` and `=` in the input can form the operator token `-=` (§3.12) only if there is no intervening white space or comment.

As a special concession for compatibility with certain operating systems, the ASCII SUB character (`\u001a`, or control-Z) is ignored if it is the last character in the escaped input stream.

3.6 White Space

White space is defined as the ASCII space, horizontal tab, and form feed characters as well as line terminators (§3.4).

3.7 Comments

Java defines three kinds of *comments*:

/* *text* */ A traditional comment: all the text from /* to */ is ignored (as in C).

// *text* A single-line comment: all the text from // to the end of the line is ignored (as in C++).

/** *documentation* */A documentation comment: the enclosed text can be processed by a separate tool to prepare automatically generated documentation of the following class, interface, constructor or member (method or field) declaration. See § 17 for a full description how the *documentation* is processed.

These comments are formally specified by the following lexical grammar:

Comment:
  / * *NotStar CommentTail*
  / / *CharactersInLine*$_{opt}$ *LineTerminator*
  / * * *DocCommentTail*

NotStar:
  InputCharacter, but not *
  LineTerminator

CommentTail:
  * *CommentTailStar*
  NotStar CommentTail

CommentTailStar:
  /
  * *CommentTailStar*
  NotStar CommentTail

CharactersInLine:
  InputCharacter
  CharactersInLine InputCharacter

j.12345.doc 16 Wed Mar 27 08:55:12 1996

DocCommentTail:
    CommentTailStar

The grammar implies all of the following properties:

- Comments do not nest.
- /* and */ have no special meaning in // comments.
- // has no special meaning in comments that begin with /* or /**.

As a result, the text:

```
/* this comment /* // /** ends here: */
```

is a single complete comment.

The lexical grammar implies that comments do not occur within character literals (§3.10.4) or string literals (§3.10.5).

3.8 Keywords

The following character sequences, formed from ASCII letters, are reserved for use as *keywords*, and are specifically not legal identifiers (§3.9):

Keyword: one of

abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	
const	for	new	synchronized	
continue	goto	package	this	

The keywords const and goto are reserved by Java, even though they are not currently used in Java. This may allow a Java compiler to produce better error messages if these C++ keywords are incorrectly used in Java programs.

While true and false might appear to be keywords, they are technically Boolean literals (§3.10.3); and while null might appear to be a keyword, it is technically the null literal (§3.10.7).

3.9 Identifiers

An *identifier* is an unlimited length sequence of Unicode *letters* and *digits*, the first of which must be a letter.

Identifier:
IdentifierName, but not a *Keyword* or *BooleanLiteral*

IdentifierName:
JavaLetter
Identifier JavaLetterOrDigit
Identifier JavaDigit

JavaLetter:
any Unicode character that is a Java letter (see below)

JavaLetterOrDigit:
any Unicode character that is a Java letter or digit (see below)

Letters and digits may be drawn from the entire Unicode character set, which supports most writing scripts in use in the world today, including the large sets for Chinese, Japanese, and Korean. This allows Java programmers to use identifiers in their programs that are written in their native languages.

The method `Character.isJavaLetter` (§19.4.17) returns `true` when given a Unicode character that is considered to be a letter in Java identifiers; and the method `Character.isJavaLetterOrDigit` (§19.4.18) returns `true` when given a Unicode character that is considered to be a letter or digit in Java identifiers.

The letters include uppercase and lowercase ASCII Latin letters `A`–`Z` (`\u0041`–`\u005a`), and `a`–`z` (`\u0061`–`\u007a`), and, for historical reasons, the ASCII underscore (`_`, or `\u005f`) and dollar sign (`$`, or `\u0024`). The digits include the ASCII digits `0-9` (`\u0030`–`\u0039`).

An identifier must not have the same spelling (Unicode character sequence) as a keyword (§3.8) or a Boolean literal (§3.10.3).

Two identifiers are the same only if they are identical, that is, have the same Unicode character for each letter or digit.

Identifiers that have the same external appearance may yet be different. For example, the identifiers consisting of the single letters LATIN CAPITAL LETTER A (`A`, `\u0041`), LATIN SMALL LETTER A (`a`, `\u0061`), GREEK CAPITAL LETTER ALPHA (`A`, `\u0391`), and CYRILLIC SMALL LETTER A (`a`, `\u0430`) are all different.

Unicode composite characters are different from the decomposed characters. For example, a LATIN CAPITAL LETTER A ACUTE (`Á`, `\u00c1`) could be considered to be the same as a LATIN CAPITAL LETTER A (`A`, `\u0041`) followed by a NON-SPACING

ACUTE (´, \u0301) when sorting, but these are different in Java identifiers. See *The Unicode Standard*, Volume 1, pages 412ff for details about decomposition, and pages 626–627 for details about sorting.

Examples of identifiers are:

```
String      i3      αρετε      MAX_VALUE     isLetterOrDigit
```

3.10 Literals

A *literal* is the source code representation of a value of a primitive type (§4.2) or the String type (§4.3.3, §19.11) or the null type (§4.1):

Literal:
  IntegerLiteral
  FloatingPointLiteral
  BooleanLiteral
  CharacterLiteral
  StringLiteral
  NullLiteral

3.10.1 Integer Literals

See §4.2.1 for a general discussion of the integer types and values.

Integer literals may be expressed in decimal (base 10), hexadecimal (base 16), or octal (base 8):

IntegerLiteral:
  DecimalLiteral IntegerTypeSuffix$_{opt}$
  HexLiteral IntegerTypeSuffix$_{opt}$
  OctalLiteral IntegerTypeSuffix$_{opt}$

IntegerTypeSuffix: one of
  l L

An integer literal is of type long if it is suffixed with a letter L or l (ell); otherwise it is of type int (§4.2.1). The suffix L is preferred, because the letter l (ell) is often hard to distinguish from the digit 1 (one).

A decimal literal consists of a digit from 1 to 9, optionally followed by one or more digits from 0 to 9, and represents a positive integer:

DecimalLiteral:
  0
  NonZeroDigit Digits$_{opt}$

Digits:
Digit
Digits Digit

Digit:
0
NonZeroDigit

NonZeroDigit: one of
1 2 3 4 5 6 7 8 9

A hexadecimal literal consists of a leading 0x or 0X followed by one or more hexadecimal digits and can represent a positive, zero, or negative integer. Hexadecimal digits with values 10 through 15 are represented by the letters a through f or A through F, respectively; each letter used as a hexadecimal digit may be uppercase or lowercase.

HexLiteral:
0x *HexDigit*
0X *HexDigit*
HexLiteral HexDigit

HexDigit: one of
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

An octal literal consists of a digit 0 followed by one or more of the digits 0 through 7 and can represent a positive, zero, or negative integer.

OctalLiteral:
0 *OctalDigit*
OctalLiteral OctalDigit

OctalDigit: one of
0 1 2 3 4 5 6 7

The largest decimal literal of type int is 2147483648 (2^{31}). All decimal literals from 0 to 2147483647 may appear anywhere an int literal may appear, but the literal 2147483648 may appear only as the operand of the unary negation operator -.

The largest positive hexadecimal and octal literals of type int are 0x7fffffff and 017777777777, respectively, both representing 2147483647 ($2^{31} - 1$). The most negative hexadecimal and octal literals of type int are 0x80000000 and 020000000000 respectively, each of which represents the decimal value –2147483648 (-2^{31}). The hexadecimal and octal literals 0xffffffff and 037777777777, respectively, represent the decimal value -1.

A compile-time error occurs if a decimal literal of type int is larger than 2147483648 (2^{31}), or if the literal 2147483648 appears anywhere other than as the operand of the unary - operator, or if a hexadecimal or octal int literal does not fit in 32 bits.

Examples of int literals:

```
0       2       0372        0xDadaCafe          1996    0x00FF00FF
```

The largest decimal literal of type long is 9223372036854775808L (2^{63}). All decimal literals from 0L to 9223372036854775807L may appear anywhere a long literal may appear, but the literal 9223372036854775808L may appear only as the operand of the unary negation operator -.

The largest positive hexadecimal and octal literals of type long are 0x7fffffffffffffffL and 0777777777777777777777L, respectively; each represents 9223372036854775807L ($2^{63} - 1$). The literals 0x8000000000000000L and 0400000000000000000000L are the most negative long hexadecimal and octal literals respectively. Each has the decimal value –9223372036854775808L (-2^{63}). The hexadecimal and octal literals 0xffffffffffffffffL and 01777777777777777777777L, respectively, represent the decimal value -1L.

A compile-time error occurs if a decimal literal of type long is larger than 9223372036854775808L (2^{63}), or if the literal 9223372036854775808L appears anywhere other than as the operand of the unary - operator, or if a hexadecimal or octal long literal does not fit in 64 bits.

Examples of long literals:

```
0l      0777L       0x100000000L        2147483648L     0xC0B0L
```

3.10.2 Floating-Point Literals

See §4.2.3 for a general discussion of the floating-point types and values.

A floating-point literal has the following parts: a whole-number part, a decimal point, a fractional part, an exponent, and a type suffix. The exponent, if present, is indicated by a letter e or E followed by an optionally signed integer.

At least one digit, in either the whole number or the fraction part, and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional.

A floating-point literal is of type float if it is suffixed with a letter F or f; otherwise its type is double, and can optionally be suffixed with D or d.

FloatingPointLiteral:
  Digits . Digits$_{opt}$ *ExponentPart*$_{opt}$ *FloatTypeSuffix*$_{opt}$
  . Digits ExponentPart$_{opt}$ *FloatTypeSuffix*$_{opt}$
  Digits ExponentPart FloatTypeSuffix$_{opt}$
  Digits ExponentPart$_{opt}$ *FloatTypeSuffix*

ExponentPart:
  ExponentIndicator SignedInteger

ExponentIndicator: one of
  e E

SignedInteger:
  Sign$_{opt}$ *Digits*

Sign: one of
  + -

FloatTypeSuffix: one of
  f F d D

The Java types float and double are IEEE 754 32-bit single-precision and 64-bit double-precision binary floating-point values, respectively.

The details of proper input conversion from a Unicode string representation of a floating-point number to the internal IEEE 754 binary floating-point representation are described for the methods valueOf for class Double (§19.9.16) and Float (§19.8.17) of the package java.lang.

The largest positive finite literal of type float is 3.40282347e+38f. The smallest positive finite literal of type float is 1.40239846e-45f. The largest positive finite literal of type double is 1.79769313486231570e+308. The smallest positive finite literal of type double is 4.94065645841246544e-324.

A compile-time error occurs if a nonzero floating-point literal is too large, so that on rounded conversion to its internal representation it becomes an IEEE 754 infinity. A compile-time error occurs if a nonzero floating-point literal is too small, so that on rounded conversion to its internal representation it becomes a zero. A Java programs can represent infinities by using constant expressions such as 1f/0f or -1d/0d or by using the predefined constants POSITIVE_INFINITY and NEGATIVE_INFINITY of the classes Float (§19.8) and Double (§19.9).

Predefined constants representing Not-a-Number values are defined in the classes Float and Double as Float.NaN (§19.8.5) and Double.NaN (§19.9.5).

Examples of float literals:

```
1e1f      2.f      .3f     0f     3.14f      6.022137e+23f
```

Examples of double literals:

```
1e1        2.        .3      0.0      3.14        1e-9d
```

There is no provision for expressing floating-point literals in other than decimal radix. However, certain methods in the classes Float (§19.8.23) and Double (§19.9.22) provide a way to express floating-point values in terms of hexadecimal or octal integer literals:

```
Float.intBitsToFloat(0x403faaaa)
```

3.10.3 Boolean Literals

The boolean type has two values: true and false.

BooleanLiteral: one of
  true false

3.10.4 Character Literals

A literal of type char (§4.2.1) is expressed as a character or an escape sequence, enclosed in single quotes. (The single-quote, or apostrophe, character is \u0027.)

CharacterLiteral:
  ' *SingleCharacter* '
  ' *EscapeSequence* '

SingleCharacter:
  InputCharacter, but not ' or \

The escape sequences are described in §3.10.6.

As specified in §3.4, the characters CR and LF are never an *InputCharacter*; they are recognized as constituting a *LineTerminator*.

It is a compile-time error for the character following the *SingleCharacter* or *Escape* to be other than a '. It is a compile-time error for a line terminator to appear after the opening ' and before the closing '.

Because Unicode escapes are processed very early, it is not correct to write '\u000a' for a character literal whose value is linefeed (LF); the Unicode escape \u000a is transformed into an actual linefeed in translation step 1 (§3.3), the linefeed becomes a *LineTerminator* in step 2 (§3.4), and so the character literal is not valid in step 3. Instead, one should use the escape sequence (§3.10.6) '\n'. Similarly, it is not correct to write '\u000d' for a character literal whose value is carriage return (CR). Instead, use '\r'.

Examples of char literals:

```
'a'        '\t'        '\\'        '\u03a9'        '\177'
```

3.10.5 String Literals

A string literal is zero or more characters enclosed in double quotes.

StringLiteral:
    " *StringCharacters*$_{opt}$ "

StringCharacters:
    StringCharacter
    StringCharacters StringCharacter

StringCharacter:
    InputCharacter, but not " or \
    EscapeSequence

The escape sequences are described in §3.10.6.

As specified in §3.4, neither of the characters CR and LF is ever considered to be an *InputCharacter*; each is recognized as constituting a *LineTerminator*.

It is a compile-time error for a line terminator to appear after the opening " and before the closing matching ". A long string literal can always be broken up into shorter pieces and written as a (possibly parenthesized) expression using the string concatenation operator + (§14.17.1).

Because Unicode escapes are processed very early, it is not correct to write "\u000a" for a string literal containing a single linefeed (LF); the Unicode escape \u000a is transformed into an actual linefeed in translation step 1 (§3.3), the linefeed becomes a *LineTerminator* in step 2 (§3.4), and so the string literal is not valid in step 3. Instead, one should write "\n" (§3.10.6). Similarly, it is not correct to write "\u000d" for a string literal containing a single carriage return (CR). Instead use "\r".

Examples of string literals:

```
""                              // The empty string
"\""                            // A string containing " alone
"This is a string"              // A string containing 16 characters

"This is a " +                  // Actually a string-valued expression
    "two-line string"           //      containing two string literals
```

Each string literal is a reference (§4.3) to an instance (§12.5) of class String (§19.11). String objects have a constant value. A string literal always references the same instance of class String. Two string literals reference the same instance of String if and only if they are exactly the same sequence of Unicode characters. String literals—or, more generally, strings that are the values of constant expressions (§14.27)—are “interned” so as to share unique instances, using the method String.intern (§19.11.47). Thus the test program:

```
class Other { static String hello = "Hello"; }

class Test {
    public static void main(String args[]) {
        String hello = "Hello";
        String lo = "lo";
        System.out.print((Other.hello == hello) + " ");
        System.out.print((hello == ("Hel"+"lo")) + " ");
        System.out.print((hello == ("Hel"+lo)) + " ");
        System.out.println(hello == ("Hel"+lo).intern());
    }
}
```

produces the output:

```
true true false true
```

This example illustrates four points:

- Literal strings from different classes are the same.
- Strings computed by constant expressions are computed at compile time and then treated as if they were literals.
- Strings computed at run time are newly created and therefore distinct.
- The result of explicitly interning a computed string is the same string as any pre-existing literal string with the same contents.

3.10.6 Escape Sequences for Character and String Literals

The character and string *escape sequences* allow for the representation of some nongraphic characters as well as the single quote, double quote, and backslash characters in character literals (§3.10.4) and string literals (§3.10.5).

EscapeSequence:

```
    \ b             /* \u0008: backspace BS                  */
    \ t             /* \u0009: horizontal tab HT             */
    \ n             /* \u000a: linefeed LF                   */
    \ f             /* \u000c: form feed FF                  */
    \ r             /* \u000d: carriage return CR            */
    \ "             /* \u0022: double quote                  */
    \ '             /* \u0027: single quote                  */
    \ \             /* \u005c: backslash \                   */
    OctalEscape     /* \u0000 to \u00ff: from octal value    */
```

OctalEscape:
  \ *OctalDigit*
  \ *OctalDigit OctalDigit*
  \ *ZeroToThree OctalDigit OctalDigit*

OctalDigit: one of
```
0 1 2 3 4 5 6 7
```

ZeroToThree: one of
```
0 1 2 3
```

It is a compile-time error if the character following a backslash in an escape is not `b`, `t`, `n`, `f`, `r`, `"`, `'`, `\`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, or `7`. The Unicode escape `\u` is processed earlier (§3.3).

Note that octal escapes can express only Unicode values `\u0000` through `\u00FF`. They are provided for compatibility with C and C++. However, Unicode escapes are to be preferred for most purposes.

3.10.7 The Null Literal

The null type has one value, the null reference, denoted by the literal `null`.

NullLiteral:
```
null
```

3.11 Separators

The following characters are used in Java as *separators* (punctuators):

Separator: one of
```
(    )    {    }    [    ]    ;    ,    .
```

3.12 Operators

The following tokens are used in Java as *operators*:

Operator: one of
```
=    >    <    !    ~    ?    :
==   <=   >=   !=   &&   ||   ++   --
+    -    *    /    &    |    ^    %    <<   >>   >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=  >>=  >>>=
```

CHAPTER 4

Types, Values, and Variables

Oft on the dappled turf at ease
I sit, and play with similes
Loose types of things through all degrees.
– Wordsworth

JAVA is a *strongly typed* language, which means that every variable and every expression has a type that is known at compile time. Types limit the values that a variable (§4.6) can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong typing helps detect errors at compile time.

The types of the Java language are divided into two categories: *primitive types* and *reference types*. The primitive types (§4.2) are the boolean type and the numeric types. The numeric types are the integral types byte, short, int, long, and char, and the floating-point types float and double. The reference types (§4.3) are class types, interface types, and array types. An object (§4.3.1) in Java is a dynamically created instance of a class type or a dynamically created array. The values of a reference type are references to objects. All objects, including arrays, support the methods of class Object (§4.3.2). String literals are represented by String objects (§4.3.3).

Types are the same (§4.4) if they have the same fully qualified names. Names of types are used (§4.5) in declarations, in casts, in creation expressions, and in type-testing instanceof operators.

A variable (§4.6) is a storage location. A variable of a primitive type always holds a value of that exact type. A variable of a class type *T* can hold a reference to an instance of class *T* or of any class that is a subclass of *T*. A variable of an interface type can hold a reference to any instance of any class that implements the interface. A variable of type "array of *T*" can hold a reference to any array of type "array of *S*" such that type *S* is assignable (§5.2) to type *T*. A variable of type Object can hold a reference to any object, whether class instance or array.

4.1 Primitive Types and Reference Types

There are two kinds of *types* in Java: primitive types (§4.2) and reference types (§4.3). There are, correspondingly, two kinds of data values that can be stored in variables, passed as arguments, returned by methods, and operated on: primitive values (§4.2) and reference values (§4.3).

Type:
    PrimitiveType
    ReferenceType

There is also a special *null type*, the type of the expression `null`, which has no name. (Because the null type has no name, it is impossible to declare a variable of the null type or to cast to the null type.) The null reference is the only possible value of an expression of null type, and can always be converted to any reference type. In practice, the Java programmer can ignore the null type and just pretend that `null` is merely a special literal that can be of any reference type.

4.2 Primitive Types and Values

A *primitive type* is predefined by the Java language, and named by its reserved keyword (§3.8):

PrimitiveType:
    NumericType
    `boolean`

NumericType:
    IntegralType
    FloatingPointType

IntegralType: one of
    `byte short int long char`

FloatingPointType: one of
    `float double`

Primitive values do not share state with other primitive values. A variable whose type is a primitive type always holds a primitive value of the same kind. The value of a variable of primitive type can be changed only by assignment operations on that variable.

The *numeric types* are the integral types and the floating-point types.

The *integral types* are byte, short, int, and long, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively, and char, whose values are 16-bit unsigned integers representing Unicode characters.

The *floating-point types* are float, whose values are 32-bit IEEE 754 floating-point numbers, and double, whose values are 64-bit IEEE 754 floating-point numbers.

The boolean type has the truth values true and false.

4.2.1 Integral Types and Values

The values of the integral types are integers in the following ranges:

- For byte, from –128 to 127, inclusive
- For short, from 32768 to 32767, inclusive
- For int, from –2147483648 to 2147483647, inclusive
- For long, from –9223372036854775808 to 9223372036854775807, inclusive
- For char, from '\u0000' to '\uffff' inclusive, that is, from 0 to 65535

4.2.2 Integer Operations

Java provides a number of operators that act on integral values, namely:

- The comparison operators, which result in a value of type boolean:
  - The numerical comparison operators <, <=, >, and >= (§14.19.1)
  - The numerical equality operators == and != (§14.20.1)
- The numerical operators, which result in a value of type int or long:
  - The unary plus and minus operators + and - (§14.14.3, §14.14.4)
  - The multiplicative operators *, /, and % (§14.16)
  - The additive operators + and - (§14.17.2)
  - The increment operator ++, both prefix (§14.14.1) and postfix (§14.13.1)
  - The decrement operator --, both prefix (§14.14.2) and postfix (§14.13.2)
  - The signed and unsigned shift operators <<, >>, and >>> (§14.18)
  - The bitwise complement operator ~ (§14.14.5)
  - The integer bitwise operators &, |, and ^ (§14.21.1)
- The conditional operator ? : (§14.24)
- The cast operator, which can convert from an integral value to a value of any

specified numeric type (§5.4, §14.15)

- The string concatenation operator + (§14.17.1), which, when given a String operand and an integral operand, will convert the integral operand to a String representing its value in decimal form, and then produce a newly created String that is the concatenation of the two strings

Other useful constructors, methods, and constants are predefined in the classes Integer (§19.6), Long (§19.7), and Character (§19.4).

If a numerical operator other than a shift operator has at least one operand of type long, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type long. If the other operand is not long it is first widened (§5.2) to type long by numeric promotion (§5.6). Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type int. If the other operand is not an int it is first widened to type int by numeric promotion.

The built-in integer operators do not indicate overflow or underflow in any way. The only numeric operators that can throw an exception (§11) are the integer divide operator / (§14.16.2) and the integer remainder operator % (§14.16.3), which throw an ArithmeticException if the right-hand operand is zero.

The example:

```
class Test {
	public static void main(String args[]) {
		int i = 1000000;
		System.out.println(i * i);
		long l = i;
		System.out.println(l * l);
		System.out.println(20296/(l - i));
	}
}
```

produces the output:

```
-727379968
1000000000000
```

and then encounters an ArithmeticException in the division by l-i, since l-i is zero. The first multiplication is performed in 32-bit precision, whereas the second multiplication is a long multiplication. The value -727379968 is the decimal value of the low 32 bits of the mathematical result, 1000000000000, which is a value too large for type int.

Any value of any integral type may be cast to or from any numeric type. There are no casts between integral types and the type boolean.

4.2.3 Floating-Point Types and Values

The floating-point types are float and double, representing single-precision 32-bit and double-precision 64-bit format IEEE 754 values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

The IEEE 754 standard includes not only positive and negative sign-magnitude numbers, but also positive and negative zeros, positive and negative *infinities*, and a special *Not-a-Number* (hereafter abbreviated NaN). The NaN value is used to represent the result of certain operations such as dividing zero by zero. NaN constants of both float and double type are predefined as Float.NaN (§19.8.5) and Double.NaN (§19.9.5).

The finite nonzero values of type float are of the form $s \cdot m \cdot 2^e$, where s is +1 or –1, m is a positive integer less than 2^{24}, and e is an integer between –149 and 104, inclusive.

The finite nonzero values of type double are of the form $s \cdot m \cdot 2^e$, where s is +1 or –1, m is a positive integer less than 2^{53}, and e is an integer between –1075 and 970, inclusive.

Positive zero and negative zero compare equal; for example, the result of the expression 0.0==-0.0 istrue and the result of 0.0>-0.0 is false. But there are other operations that can distinguish positive and negative zero; for example, 1.0/0.0 produces positive infinity while 1.0/-0.0 produces negative infinity.

Except for NaN, floating-point values are *ordered*; arranged from smallest to largest, they are negative infinity, negative finite values, negative zero, positive zero, positive finite values, and positive infinity.

NaN is *unordered*, so the numerical comparison operators <, <=, >, and >= return false if either or both of their operands are NaN (§14.19.1). The numerical equality operator == returns false if either operand is NaN, and the inequality operator != returns true if either operand is NaN (§14.20.1). In particular, x==x is false if and only if x is NaN, and (x<y)==!(x>=y) will be false if x or y is NaN.

Any value of a floating-point type may be cast to or from any numeric type. There are no casts between floating-point types and the type boolean.

4.2.4 Floating-Point Operations

Java provides a number of operators that act on floating-point values, namely:

- The comparison operators, which result in a value of type boolean:
  - The numerical comparison operators <, <=, >, and >= (§14.19.1)
  - The numerical equality operators == and != (§14.20.1)

- The numerical operators, which result in a value of type float or double:
  - The unary plus and minus operators + and - (§14.14.3, §14.14.4)
  - The multiplicative operators *, /, and % (§14.16)
  - The additive operators + and - (§14.17)
  - The increment operator ++, both prefix (§14.14.1) and postfix (§14.13.1)
  - The decrement operator --, both prefix (§14.14.2) and postfix (§14.13.2)
- The conditional operator ? : (§14.24)
- The cast operator, which can convert from a floating-point value to a value of any specified numeric type (§5.4, §14.15)
- The string concatenation operator + (§14.17.1)(§14.17.1), which, when given a String operand and a floating-point operand, will convert the floating-point operand to a String representing its value in decimal form, and then produce a newly created String that is the concatenation of the two strings

Other useful constructors, methods, and constants are predefined in the classes Float (§19.8), Double (§19.9), and Math (§19.10).

If at least one of the operands to a binary operator is of floating-point type, then operation is a floating-point operation, even if the other operand is integral.

If at least one of the operands to a numerical operator is of type double, then the operation is carried out using 64-bit floating-point arithmetic, and the result of the numerical operator is a value of type double (if the other operand is not a double it is first widened to type double by numeric promotion (§5.6)). Otherwise, the operation is carried out using 32-bit floating-point arithmetic, and the result of the numerical operator is a value of type float. If the other operand is not a float, it is first widened to type float by numeric promotion.

Operators on floating-point numbers behave exactly as specified by IEEE 754. In particular, Java requires support of IEEE 754 *denormalized* floating-point numbers and *gradual underflow*, which make it easier to prove desirable properties of particular numerical algorithms.

Java requires that floating-point arithmetic behave as if every floating-point operator rounded its floating-point result to the result precision. *Inexact* results must be rounded to the representable value nearest to the infinitely precise result; if the two nearest representable values are equally near, the one with its least significant bit zero is chosen. This is the IEEE 754 standard's default rounding mode known as *round to nearest*.

Java uses *round toward zero* when converting a floating value to an integer (§5.1.3), which acts, in this case, as though the number were truncated, discarding

j.12345.doc 33 Wed Mar 27 08:55:12 1996

the mantissa bits. Round toward zero chooses at its result the format's value closest to and no greater in magnitude than the infinitely precise result.

Java floating-point operators produce no exceptions (§11). An operation that overflows produces a signed infinity, an operation that underflows produces a signed zero, and an operation that has no mathematically definite result produces NaN. All numeric operations with NaN as an operand produce NaN as a result. As has already been described, NaN is unordered, so numeric comparison operations involving one or two NaNs are `false` and any `!=` comparison involving NaN returns `true`, including `x!=x` when `x` is NaN.

The example program:

```
class Test {
	public static void main(String args[]) {
		// an example of overflow:
		double d = 1e308;
		System.out.print("overflow produces infinity: ");
		System.out.println(d + "*10==" + d*10);

		// an example of gradual underflow:
		System.out.print("gradual underflow:");
		float f = 1e-34f;
		for (int i = 0; i < 4; i++)
			System.out.print(" " + (f /= 1000));
		System.out.println();

		// an example of NaN:
		System.out.print("0.0/0.0 is Not-a-Number: ");
		d = 0.0/0.0;
		System.out.println(d);

		// an example of inexact results and rounding:
		System.out.print("inexact results with float:");
		for (int i = 0; i < 100; i++) {
			float z = 1.0f/i;
			if (z*i != 1.0f)
				System.out.print(" " + i);
		}
		System.out.println();
		System.out.print("inexact results with double:");
		for (int i = 0; i < 100; i++) {
			double z = 1.0/i;
			if (z*i != 1.0)
				System.out.print(" " + i);
		}
		System.out.println();

		// an example of cast to integer rounding:
		System.out.print("cast to int rounds toward 0: ");
		d = 12345.6;
```

```
                System.out.println((int)d + " " + (int)(-d));
            }
        }
```

produces the output:

```
overflow produces infinity: 1e+308*10==Inf
gradual underflow: 1e-37 9.99995e-41 9.94922e-44 0
0.0/0.0 is Not-a-Number: NaN
inexact results with float: 0 41 47 55 61 82 83 94 97
inexact results with double: 0 49 98
cast to int rounds toward 0: 12345 -12345
```

Bug in 1.0: The output format for floating-point number shown here reflects the use of %g printf format, not the language-specified output format described in §19.8.9 and §19.9.8.

This example demonstrates, among other things, that gradual underflow involves a gradual loss of precision.

The inexact results when i is 0 involve division by zero, so that z becomes positive infinity, and z*0 is NaN, which is not equal to 1.0.

4.2.5 The boolean Type and its Values

The boolean type represents a logical quantity with two possible values, indicated by the literals true and false (§3.10.3). The boolean operators are:

- The relational operators == and != (§14.20.2)
- The logical operator ! (§14.14.6)
- The logical operators &, ^, and | (§14.21.2)
- The conditional-and and conditional-or operators && (§14.22) and || (§14.23)
- The conditional operator ? : (§14.24)
- The string concatenation operator + (§14.17.1), which, when given a String operand and a boolean operand, will convert the boolean operand to a String (either "true" or "false"), and then produce a newly created String that is the concatenation of the two strings

Boolean expressions control the control flow in:

- The if statement (§13.8)
- The while statement (§13.10)
- The do statement (§13.11)
- The for statement (§13.12)

and determine which subexpression is chosen to be evaluated in the conditional ? : operator (§14.24).

Only boolean expressions can be used in the control flow statements and as the first operand of the conditional operator ? :. An integer x can be converted to a boolean, following the C language convention that any nonzero value is true, by the expression x!=0. An object reference obj can be converted to a boolean, following the C language convention that any reference other than null (§14.7.2) is true, by the expression obj!=null.

There are no casts between the type boolean and any other type, although a cast of a boolean value to type boolean is allowed (§5.5.1).

4.3 Reference Types, Objects, and Reference Values

There are three kinds of *reference types*: the class types (§8), the interface types (§9), and the array types (§10).

```
ReferenceType:
    ClassOrInterfaceType
    ArrayType

ClassOrInterfaceType:
    TypeName

ArrayType:
    PrimitiveType [ ]
    TypeName [ ]
    ArrayType [ ]
```

Names, and specifically type names, are described in §6.

The sample code:

```
class Point { int[] metrics; }
interface Move { void move(int deltax, int deltay); }
```

declares a class type Point, an interface type Move, and uses an array type int[] (an array of int) to declare the field metrics of the class Point.

4.3.1 Objects

An *object* is a dynamically created class instance or a dynamically created array.

The reference values (often just *references*) are *pointers* to these objects, and a special null reference, which refers to no object.

A class instance is explicitly created by a class instance creation expression (§14.8), or by invoking the newInstance method of class Class (§19.2.7). An array is explicitly created by an array creation expression (§14.8).

A new class instance is implicitly created when the string concatenation operator + (§14.17.1) is used in an expression, resulting in a new object of type String (§4.3.3, §19.11). A new array object is implicitly created when an array initializer expression (§10.6) is evaluated, either at class or interface load time (§12.2), when a new instance of a class is created (§14.8), or when a local variable declaration statement is executed (§13.3).

Many of these cases are illustrated in the following example:

```
class Point {
	int x, y;
	// Point instance explicitly created at class load time:
	static Point origin = new Point(0,0);
	Point() { System.out.println("default"); }
	Point(int x, int y) { this.x = x; this.y = y; }
	// String implicitly created by + operator:
	public String toString() { return "(" + x + "," + y + ")"; }
}

class Test {
	public static void main(String args[]) {
		// Point explicitly created using newInstance:
		Point p = null;
		try {
			p = (Point)Class.forName("Point").newInstance();
		} catch (Exception e) {
			System.out.println(e);
		}
		// array implicitly created by array constructor:
		Point a[] = { new Point(0,0), new Point(1,1) };
		// Strings implicitly created by + operators below...
		System.out.println("p: " + p);
		System.out.println("a: { " + a[0] + ", " + a[1] + " }");
		// array explicitly created by array creation expression:
		String sa[] = new String[2];
		sa[0] = "he"; sa[1] = "llo";
		System.out.println(sa[0] + sa[1]);
	}
}
```

The operators on objects are:

- Field access, using either a qualified name (§6.5) or a field access expression (§14.10)
- Method invocation (§14.11)

- The cast operator (§5.4, §14.15)
- The string concatenation operator + (§14.17.1), which, when given a String operand and a reference, will convert the reference to a String by invoking the toString method (§19.1.2) of the referenced object (or using "null" if it is a null reference), and then produce a newly created String that is the concatenation of the two strings
- The instanceof operator (§14.19.2)
- The reference equality operators == and!= (§14.20.3)
- The conditional operator ? : (§14.24).

There may be many references to the same object. Most objects have state, stored in the fields of objects that are instances of classes or in the variables that are the components of an array object. If two variables contain references to the same object, the state of the object can be modified using one variable's reference to the object and then the altered state can be observed through the other variable's reference. Thus the example program:

```
class Value { int val; }

class Test {
    public static void main(String args[]) {
        int i1 = 3;
        int i2 = i1;
        i2 = 4;
        System.out.print("i1==" + i1);
        System.out.println(" but i2==" + i2);
        Value v1 = new Value();
        v1.val = 5;
        Value v2 = v1;
        v2.val = 6;
        System.out.print("v1.val==" + v1.val);
        System.out.println(" and v2.val==" + v2.val);
    }
}
```

produces the output:

```
i1==3 but i2==4
v1.val==6 and v2.val==6
```

because v1.val and v2.val reference the same instance variable (§4.6.3) in the one Value object created by the only new expression, while i1 and i2 are different variables.

See §10 and §14.9 for examples of the creation and use of arrays.

Each object has an associated lock (§16.12), which is used by synchronized methods (§8.4.4) and the synchronized statement (§13.17) to provide control over concurrent access to state by multiple threads (§16.11).

4.3.2 The Class Object

The standard class Object is a superclass (§8.1) of all other classes. A variable of type Object can hold a reference to any object, whether it is an instance of a class or an array (§10). All class and array types inherit the methods of class Object, which are described by the following method signatures (§8.4.3), and completely specified in §19.1:

```
package java.lang;

public class Object {
    public final Class getClass();
    public String toString();
    public boolean equals(Object obj);
    public int hashCode();
    protected Object clone() throws CloneNotSupportedException;
    public final void notify();
    public final void notifyAll();
    public final void wait(long timeout);
    public final void wait(long timeout, int nanos);
    public final void wait() throws InterruptedException;
    protected void finalize() throws Throwable;
}
```

The members of Object are as follows:

- The method getClass returns the Class object that represents the class of the object. A Class object exists for each class type and array type, and can be used to discover the fully qualified name of the class, its members, its immediate superclass, and any interfaces that it implements. A class method that is declared synchronized (§8.4.4.5) synchronizes on the lock associated with the Class object of the class.
- The method toString returns a String representation of the contents of the object, suitable for printing.
- The methods equals and hashCode are declared for the benefit of hash tables such as java.util.Hashtable (§20.7). The method equals defines a notion of object equality, based on value, not reference.
- The method clone is used to make a duplicate of an object and is described in TBD.X.
- The methods notify, notifyAll, and wait are used in concurrent programming us-

ing threads, as described in §16.

- The method finalize is run just before an object is destroyed and is described in §12.6.

4.3.3 The Class String

Instances of class String (§19.11) represent sequences of Unicode characters. A String object has a constant, unchanging value. String literals (§3.10.5) are references to instances of class String.

The string concatenation + operator (§14.17.1) implicitly creates a new String object, implicitly converting any single non-String operand to a String.

4.4 When Class Types Are the Same

Two classes are the *same class* (and therefore the *same type*) if they are loaded by the same class loader and they have the same fully qualified name (§6.5).

4.5 Where Types Are Used

Types are used when they appear in declarations or in certain expressions. The following code fragment contains an instance of each kind of usage of a type:

```
import java.util.Random;

class MiscMath {
    int divisor;
    float ratio(long l) {t
        float f;
        try {
            l /= divisor;
        } catch (Exception e) {
            if (e instanceof ArithmeticException)
                l = Long.MAX_VALUE;
            else
                l = 0;
        }
        return (float)l;
    }
    double gausser() {
        Random r = new Random();
        double[] val = new double[2];
        val[0] = r.nextGaussian();
        val[1] = r.nextGaussian();
        return (val[0]+val[1])/2;
        }
        MiscMath(long by) { divisor = (int)by; }
}
```

Types are used in declarations of:

- Imported types (§7.5); here the type Random, imported from the type java.util.Random of the package java.util
- Fields, which are the class variables and instance variables of classes (§8.3), and constant values of interfaces (§9.3); here the field divisor in the class MiscMath is declared using type int
- Method parameters (§8.6.1); here the parameter l of the method ratio is declared with type long
- Method results (§8.4); here the result of the method ratio is declared to have type float, and the result of the method gausser is of type double
- Constructor parameters (§8.6.1); here the parameter of the constructor for MiscMath is declared with type long
- Local variables (§13.3, §13.12); here the local variable f of the method ratio is declared with type float, and the locals r and val of the method gausser are declared with types Random and double[] (array of double)
- Exception handler parameters (§13.18); here the exception handler parameter e of the catch clause is declared with type Exception

and in expressions in:

- Class instance creations (§14.8); here a local variable r of method gausser is initialized by a class instance creation expression that uses the type Random
- Array creations (§14.9); here the local variable val is declared with type array of double, and initialized by an array creation expression that creates an array of double of size 2
- Casts (§14.15); here the return statement of the method ratio uses the float type in a cast
- The instanceof operator (§14.19.2); here the instanceof operator tests whether e is assignment compatible with the type ArithmeticException

4.6 Variables

A variable is a storage location and has an associated type, sometimes called its *compile-time type*, that is either a primitive type (§4.2) or a reference type (§4.3). A variable always contains a value that is assignment compatible (§5.2) with its type. A variable's value is changed by an assignment (§14.25) or by a prefix or postfix ++ (increment) or -- (decrement) operator (§14.13.1, §14.13.2, §14.14.1, §14.14.2).

Compatibility of the value of a variable with its type is guaranteed by the design of the Java language because default values are compatible (§4.6.4) and all assignments to a variable are checked, at compile time, for assignment compatibility (§5.2).

4.6.1 Variables of Primitive Type

A variable of a primitive type always holds a value of that exact primitive type.

4.6.2 Variables of Reference Type

A variable of reference type can hold either of the following:

- A null reference
- A reference to any object (§4.3) whose class (§4.6.5) is assignment compatible with type of the variable (§5.3.2)

4.6.3 Kinds of Variables

There are seven kinds of variables:

1. A *class variable* is a field of class types declared using the keyword static (§8.3) within a class declaration, or with or without the keyword static in an interface declaration. Class variables are created when the class is loaded (§12.2) and are initialized on creation to default values (§4.6.4). The class variable effectively ceases to exist when its class is unloaded (§12.8) after any necessary finalization of the class (§12.6) has been completed.
2. An *instance variable* is a field declared within a class declaration without using the keyword static (§8.3). If a class *T* has a field *a* that is an instance variable, then a new instance variable *a* is created and initialized to a default value (§4.6.4) as part of each newly created object of class *T* or of any class that is a subclass of *T* (§8.1.4). The instance variable effectively ceases to exist when the object of which it is a field is no longer referenced, after any necessary fi-

nalization of the object (§12.6) has been completed.

3. *Array components* are unnamed variables that are created and initialized to default values (§4.6.4) whenever a new object that is an array is created (§14.9). The array components effectively cease to exist when the array is no longer referenced. See §10 for a description of arrays.
4. *Method parameters* (§8.4.2) name argument values passed to a method. For every parameter declared in a method declaration, a new parameter variable is created each time that method is invoked (§14.11). The new variable is initialized with the corresponding argument value from the method invocation. The method parameter effectively ceases to exist when the execution of the body of the method is complete.
5. *Constructor parameters* (§8.6.1) name argument values passed to a constructor. For every parameter declared in a constructor declaration, a new parameter variable is created each time a class instance creation expression (§14.8) or explicit constructor invocation (§8.6.5) is evaluated. The new variable is initialized with the corresponding argument value from the creation expression or constructor invocation. The constructor parameter effectively ceases to exist when the execution of the body of the constructor is complete.
6. An *exception-handler parameter* variable is created each time an exception is caught by a catch clause of a try statement (§13.18). The new variable is initialized with the actual object associated with the exception (§11.3, §13.16). The exception-handler parameter effectively ceases to exist when execution of the block associated with the catch clause is complete.
7. *Local variables* are declared by local variable declaration statements (§13.3). Whenever the flow of control enters a block (§13.2) or for statement (§13.12), a new variable is created for each local variable declared in a local variable declaration statement immediately contained within that block or for statement. The local variable is not initialized, however, until the local variable declaration statement that declares it is executed. (The rules of definite assignment (§15) prevent the value of a local variable from being used before it has been initialized or otherwise assigned a value.) The local variable effectively ceases to exist when the execution of the block or for statement is complete.

   Were it not for one exceptional situation, a local variable could always be regarded as being created when its local variable declaration statement is executed. The exceptional situation involves the switch statement (§13.9), where it is possible for control to enter a block but bypass execution of a local variable declaration statement.

The following example contains several different kinds of variables:

```
class Point {
      static int numPoints;             // numPoints is a class variable
      int x, y;                         // x and y are instance variables
      int[] w = new int[10];            // w[0] is an array component
      int setX(int x) {                 // x is a method parameter
            int oldx = this.x;          // oldx is a local variable
            this.x = x;
            return oldx;
      }
}
```

4.6.4 Initial Values of Variables

Every variable in a Java program must have a value before its value is used:

- Each class variable, instance variable, or array component is initialized with a *default value* when it is created (§14.8, §14.9, §19.2.6):
  - For type byte, the default value is zero, that is, the value of (byte)0.
  - For type short, the default value is zero, that is, the value of (short)0.
  - For type int, the default value is zero, that is, 0.
  - For type long, the default value is zero, that is, 0L.
  - For type float, the default value is positive zero, that is, 0.0f.
  - For type double, the default value is positive zero, that is, 0.0d.
  - For type char, the default value is the null character, that is, '\u0000'.
  - For type boolean, the default value is false.
  - For all reference types (§4.3), the default value is null (§14.7.2).
- Each method parameter (§8.4.2) is initialized to the corresponding argument value provided by the invoker of the method (§14.11).
- Each constructor parameter (§8.6.1) is initialized to the corresponding argument value provided by an object creation expression (§14.8) or explicit constructor invocation (§8.6.5).
- An exception-handler parameter (§13.18) is initialized to the thrown object representing the exception (§11.3, §13.16).
- A local variable (§13.3, §13.12) must be explicitly given a value before it is used, by either initialization (§13.3) or assignment (§14.25), in a way that can be verified by the compiler using the rules for definite assignment (§15).

j.12345.doc 44 Wed Mar 27 08:55:12 1996

The example program:

```
class Point {
    static int npoints;
    int x, y;
    Point root;
}

class Test {
    public static void main(String args[]) {
        System.out.println("npoints=" + Point.npoints);
        Point p = new Point();
        System.out.println("p.x=" + p.x + ", p.y=" + p.y);
        System.out.println("p.root=" + p.root);
    }
}
```

prints:

```
npoints=0
p.x=0, p.y=0
p.root=null
```

illustrating the default initialization of npoints, which occurs when the class is loaded, and the default initialization of x, y, and root, which occurs when a new Point is instantiated. See §12 for a full description of all aspects of loading, linking, instantiation, and initialization.

4.6.5 Variables Have Types, Objects Have Classes

Every object belongs to some particular class. This is the class that was mentioned in th creation expression that produced the object, or the class whose class object was used to invoke the newInstance method (§19.2.6) to produce the object. This class is called *the* class of the object. An object is said to be an instance of its class and of all superclasses of its class. Sometimes the class of an object is called its “run-time type” but “class” is the more accurate term.

(Sometimes a variable or expression is said to have a “run-time type” but that is an abuse of terminology; it refers to the class of the object referred to by the value of the variable or expression at run time, assuming that the value is not null. Properly speaking, type is a compile-time notion. A variable or expression has a type; an object or array has no type, but belongs to a class.)

The type of a variable is always declared, and the type of an expression can be deduced at compile time. The type limits the possible values that the variable can hold or the expression can produce at run time. If a run-time value is a reference that is not null, it refers to an object or array that has a class (not a type), and that class will necessarily be compatible with the compile-time type.

Even though a variable or expression may have a compile-time type that is an interface type, there are no instances of interfaces. A variable or expression whose type is an interface type can reference any object whose class implements (§8.1.5) that interface.

Here is an example of creating new objects and of the distinction between the type of a variable and the class of an object:

```
public interface Colorable {
    void setcolor(byte r, byte g, byte b);
}

class Point { int x, y; }

class ColoredPoint extends Point implements Colorable {
    byte r, g, b;
    public void setcolor(byte rv, byte gv, byte bv) {
        r = rv; g = gv; b = bv;
    }
}

class Test {
    public static void main(String args[]) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        p = cp;
        Colorable c = cp;
    }
}
```

In this example:

- The local variable p of the method main of class Test has type Point and is initially assigned a reference to a new instance of class Point.
- The local variable cp similarly has as its type ColoredPoint, and is initially assigned a reference to a new instance of class ColoredPoint.
- The assignment of the value of cp to the variable p causes p to hold a reference to a ColoredPoint object. This is permitted because ColoredPoint is a subclass of Point, so the class ColoredPoint is assignment compatible with the type Point. A ColoredPoint object includes support for all the methods of a Point and has, in addition to its particular fields r, g, and b, the fields of class Point also, namely x and y.
- The local variable c has as its type the interface type Colorable, so it can hold a reference to any object whose class implements Colorable, and specifically it can hold a reference to a ColoredPoint.
- Note that an expression such as "new Colorable()" is not valid because it is not possible to create an instance of an interface, only of a class.

Every array also has a class; the method getClass (§19.1.1), when invoked for an array object, will return a class object (of class Class) that represents the class of the array. The classes for arrays have strange names that are not valid Java identifiers; for example, the class for an array of int components has the name “[I” and so the value of the expression

```
new int[10].getClass().getName()
```

is the string "[I".

CHAPTER 5

Conversions and Promotions

Thou art not for the fashion of these times,
Where none will sweat but for promotion.
– William Shakespeare, As You Like It
II, iii, 59

EVERY expression has a type that can be deduced from the structure of the expression and the types of the literals, variables, and methods mentioned in the expression. It is possible, however, to write an expression in a context where the type of the expression is not appropriate. In some cases, this leads to an error at compile time; for example, if the expression in an if statement (§13.8) has any type other than boolean, a compile-time error occurs. In other cases, the context may be able to accept a type that is related to the type of the expression; as a convenience, rather than requiring the programmer to indicate a type conversion explicitly, the Java language performs an implicit *conversion* from the type of the expression to a type acceptable for its surrounding context.

A specific conversion from type *S* to type *T* allows an expression of type S to be treated at compile time as if it had type *T* instead. In some cases this will require a corresponding action at run time to check the validity of the conversion or to translate the run-time value of the expression into a form appropriate for the new type *T*. For example:

- A conversion from type Object to type Thread will require a run-time check to make sure that the run-time value is actually an instance of class Thread or one of its subclasses; if it is not, an exception is thrown.
- A conversion from type Thread to type Object requires no run-time action; Thread is a subclass of Object, so any reference produced by an expression of type Thread is a valid reference value of type Object.
- A conversion from type int to type long requires sign-extension of a 32-bit integer value to the 64-bit long representation. No information is lost.

- A conversion from type double to type long requires a nontrivial translation from a 64-bit floating-point value to the 64-bit integer representation. Depending on the actual run-time value, information may be lost.

In every conversion context, only certain specific conversions are permitted. The specific conversions that are possible in Java are grouped for convenience of description into several broad categories:

- Identity conversions
- Widening primitive conversions
- Narrowing primitive conversions
- Widening reference conversions
- Narrowing reference conversions
- String conversions

There are then five *conversion contexts* in which Java expressions can occur. Each context allows conversions in some of the above-named categories but not others. The term "conversion" is also used to describe the process of choosing a specific conversion for such a context. For example, we say that an expression that is an actual argument in a method invocation is subject to "method invocation conversion," meaning that a specific conversion will be implicitly chosen for that expression according to the rules for the method invocation argument context.

One conversion context is the operand of a numeric operator such as + or *. The conversion process for such operands is called *numeric promotion*. Promotion is special in that, in the case of binary operators, the conversion chosen for one operand may depend in part on the type of the other operand expression.

This chapter first describes the six categories of conversions (§5.1), including the special conversions to String allowed for the string concatenation operator +. Then the five conversion contexts are described:

- Assignment conversion (§5.2, §14.25) converts the type of an expression to the type of a specified variable. The conversions permitted for assignment are limited in such a way that assignment conversion never causes an exception.
- Method invocation conversion (§5.3, §14.8, §14.11) is applied to each argument in a method or constructor invocation, and, except in one case, performs the same conversions that assignment conversion does. Method invocation conversion never causes an exception.
- Casting conversion (§5.4) converts the type of an expression to a type explicitly specified by a cast operator (§14.15). It is more inclusive than assignment

or method invocation conversion, allowing any specific conversion other than a string conversion, but certain casts to a reference type may cause an exception at run time.

- String conversion (§14.17.1) allows any type to be converted to type `String`.
- Numeric promotion (§5.6) brings the operands of a numeric operator to a common type so that an operation can be performed.

Here are some examples of various contexts for conversion:

```
class Test {
    public static void main(String args[]) {
        // Casting conversion (§5.4) of a float literal to
        // type int. Without the cast operator, this would
        // be a compile-time error, because t his is a
        // narrowing conversion (§5.1.3):
        int i = (int)12.5f;
        // String conversion (§5.5) of i's int value:
        System.out.println("(int)12.5f==" + i);

        // Assignment conversion (§5.2) of i's value to type
        // float. This is a widening conversion (§5.1.2):
        float f = i;
        // String conversion of f's float value:
        System.out.println("after float widening: " + f);

        // Numeric promotion (§5.6) of i's value to type
        // float. This is a binary numeric promotion.
        // After promotion, the operation is float*float:
        System.out.print(f);
        f = f * i;
        // Two string conversions of i and f:
        System.out.println("*" + i + "==" + f);

        // Method invocation conversion (§5.3) of f's value
        // to type double, needed because the method Math.sin
        // accepts only a double argument:
        double d = Math.sin(f);
        // Two string conversions of f and d:
        System.out.println("Math.sin(" + f + ")==" + d);
    }
}
```

which produces the output:

```
(int)12.5f==12
after float widening: 12.0
12.0*12==144.0
Math.sin(144.0)==-0.491022// DIGITS TBD
```

5.1 Kinds of Conversion

Specific type conversions in Java are divided into six categories.

5.1.1 Identity Conversions

A conversion from a type to that same type is permitted for any type. This may seem trivial, but it has two practical consequences. First, it is always permitted for an expression to have the desired type to begin with, thus allowing the simply stated rule that every expression is subject to conversion, if only a trivial identity conversion. Second, it implies that it is permitted for a program to include redundant cast operators for the sake of clarity.

The *only* permitted conversion that involves the type boolean is the identity conversion from boolean to boolean.

5.1.2 Widening Primitive Conversions

The following 19 specific conversions on primitive types are called the *widening primitive conversions*:

- byte to short, int, long, float, or double
- short to int, long, float, or double
- char to int, long, float, or double
- int to long, float, or double
- long to float or double
- float to double

Widening conversions do not lose information about the overall magnitude of a numeric value. Indeed, conversions widening from an integral type to another integral type and from float to double do not lose any information at all; the numeric value is preserved exactly. Conversion of an int or a long value to float, or of a long value to double, may result in loss of *precision*, that is, the result may lose some of the least significant bits of the value; the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode (§4.2.4).

According to this rule, a widening conversion of a signed integer value to an integral type *T* simply sign-extends the two's-complement representation of the integer value to fill the wider format. A widening conversion of a character to an integral type *T* zero-extends the representation of the character value to fill the wider format.

Despite the fact that loss of precision may occur, widening conversions among primitive types never result in a run-time exception (§11).

Here is an example of a widening conversion that loses precision:

```
class Test {
    public static void main(String args[]) {
        int big = 1234567890;
        float approx = big;
        System.out.println(big - (int)approx);
    }
}
```

which prints:

```
-46
```

thus losing precision because values of type float do not have nine significant digits.

5.1.3 Narrowing Primitive Conversions

The following 23 specific conversions on primitive types are called the *narrowing primitive conversions*:

- byte to char
- short to byte or char
- char to byte or short
- int to byte, short, or char
- long to byte, short, char, or int
- float to byte, short, char, int, or long
- double to byte, short, char, int, long, or float

Narrowing conversions may lose information about the overall magnitude of a numeric value and may also lose precision.

A narrowing conversion of a signed integer to an integral type *T* simply discards all but the *n* lowest order bits, where *n* is the number of bits used to represent type *T*. This may cause the resulting value to have a different sign than the input value.

A narrowing conversion of a character to an integral type *T* likewise simply discards all but the *n* lowest order bits, where *n* is the number of bits used to represent type *T*. This may cause the resulting value to be a negative number, even though characters represent 16-bit unsigned integer values.

A narrowing conversion of a floating-point number to an integral type *T* takes two steps:

1. In the first step the floating-point number is converted either to a long, if *T* is long; or to an int, if *T* is byte, short, char, or int as follows:
   - If the floating-point number is NaN (§4.2.3), the result of the first step of the conversion is an int or long 0.
   - Otherwise, if the floating-point number is not an infinity, the floating-point value is rounded to an integer value *V*, rounding toward zero using IEEE 754 round-toward-zero mode (§4.2.3). Then there are two cases:
     - If *U* is long, and this integer value can be represented as a long, then the result of the first step is the long *V*.
     - Otherwise, if this integer value can be represented as an int, then the result of the first step is the int value *V*.
   - Otherwise, either
     - the value must be too small (a negative value of large magnitude or negative infinity), and the result of the first step is the smallest representable value of type int or long; or
     - the value must be too large (a positive value of large magnitude or positive infinity), and the result of the first step is the largest representable value of type int or long.
2. In the second step:
   - if *T* is int or long, the result of the conversion is the result of the first step or
   - if *T* is byte, char, or short, the result of the conversion is a narrowing conversion (§5.1.3) of the result of the first step to *T*.

The example:

```
class Test {
    public static void main(String args[]) {
        float fmin = Float.NEGATIVE_INFINITY;
        float fmax = Float.POSITIVE_INFINITY;
        System.out.print("long: " + (long)fmin);
        System.out.println(".." + (long)fmax);
        System.out.print("int: " + (int)fmin);
        System.out.println(".." + (int)fmax);
        System.out.print("short: " + (short)fmin);
        System.out.println(".." + (short)fmax);
```

```
		System.out.print("char: " + (int)(char)fmin);
		System.out.println(".." + (int)(char)fmax);
		System.out.print("byte: " + (byte)fmin);
		System.out.println(".." + (byte)fmax);
	}
}
```

produces the output:

```
long: -9223372036854775808..9223372036854775807
int: -2147483648..2147483647
short: 0..-1
char: 0..65535
byte: 0..-1
```

The results for char, int, and long are unsurprising, producing the minimum and maximum representable values of the type.

The results for byte and short lose both information about the overall magnitude of a numeric value and also lose precision. The results can be understood by examining the low order bits of the minimum and maximum int. The minimum int is, in hexadecimal, 0x80000000, and the maximum int is 0x7fffffff. This explains the char and short results, which are the low 16 bits of these values, namely, '\u0000'...'\uffff' and 0...-1 respectively; and the byte results, which are the low 8 bits of these values, namely, 0x00...0xff.

The values 3.40282347e+38f and 1.40239846e-45f are the largest positive finite and smallest positive non zero values of type float, respectively.

The values 1.79769313486231570e+308 and 4.94065645841246544e-324 are the largest positive finite and smallest finite non zero values of type double, respectively.

A narrowing conversion from double to float behaves in accordance with IEEE 754. The result is correctly rounded using IEEE 754 round-to-nearest mode. A value too small to be represented as a float is converted to positive or negative zero; a value too large to be represented as a float is converted to a (positive or negative) infinity. A double NaN is always converted to a float NaN.

Despite the fact that overflow, underflow, or loss of precision may occur, narrowing conversions among primitive types never result in a run-time exception (§11).

Here is a small test program that demonstrates a number of narrowing conversions that lose information:

```
class Test {
	public static void main(String args[]) {
		// narrowing int to short loses high bits:
		System.out.println("(short)0x12345678==0x" +
			hex((short)0x12345678));
```

```
            // int value not fitting in byte loses sign:
            System.out.println("(byte)255==" + (byte)255);

            // float too big gives largest int value
            System.out.println("(int)1e20f==" + (int)1e20f);

            // NaN's converted to int yield zero:
            System.out.println("(int)NaN==" + (int)Float.NaN);

            // doubles too large for float yield infinity:
            System.out.println("(float)-1e100==" + (float)-1e100);

            // doubles too small for float underflow to zero:
            System.out.println("(float)1e-50==" + (float)1e-50);
        }
        static String hex(long i) {
            return Long.toString(i, 16);
        }
    }
```

This test program produces the following output:

```
    (short)0x12345678==0x5678
    (byte)255==-1
    (int)1e20f==2147483647
    (int)NaN==0
    (float)-1e100==-Inf
    (float)1e-50==0
```

5.1.4 Widening Reference Conversions

The following permitted conversions are called the *widening reference conversions*:

- From any class type *S* to any class type *T*, provided that *S* is a subclass of *T*. (An important special case is that there is a widening conversion to the class type Object from any other class type.)
- From any class type *S* to any interface type *K*, provided that *S* implements *K*.
- From the null type to any class type, interface type, or array type.
- From any interface type *J* to any interface type *K*, provided that *J* is a subinterface of *K*.
- From any interface type to type Object.
- From any array type to type Object.
- From any array type *SC*[] to any array type *TC*[], provided that *SC* and *TC* are

j.12345.doc 55 Wed Mar 27 08:55:12 1996

reference types and there is a permitted widening conversion from *SC* to *TC*.

Such conversions never require a special action at run time and therefore never throw an exception at run time. They consist simply in regarding a reference as having some other type in a manner that can be proved correct at compile time.

See §8 for the detailed specifications for classes, §9 for interfaces, and §10 for arrays.

5.1.5 Narrowing Reference Conversion

The following permitted conversions are called the *narrowing reference conversions*:

- From any class type *S* to any class type *T*, provided that *S* is a superclass of *T*. (An important special case is that there is a narrowing conversion from the class type Object to any other class type.)
- From any class type *S* to any interface type *K*, provided that *S* is not final and does not implement *K*. (An important special case is that there is a narrowing conversion from the class type Object to any interface type.)
- From type Object to any array type.
- From any interface type *J* to any class type *T* that is not final.
- From any interface type *J* to any class type *T* that is final, provided that *T* implements *J*.
- From any interface type *J* to any interface type *K*, provided that *J* is not a subinterface of *K* and there is no method name *m* such that *J* and *K* both declare a method named *m* with the same signature but different return types.
- From any array type *SC*[] to any array type *TC*[], provided that *SC* and *TC* are reference types and there is a permitted narrowing conversion from *SC* to *TC*.

Such conversions require a test at run time to find out whether the actual reference value is a legitimate value of the new type. If it is not, a ClassCastException is thrown.

5.1.6 String Conversion

There is a string conversion to type String from every other type, including the null type.

5.1.7 Forbidden Conversions

There is no permitted conversion from any reference type to any primitive type.

Except for the string conversions, there is no permitted conversion from any primitive type to any reference type.

There is no permitted conversion from the null type to any primitive type.

There is no permitted conversion to the null type other than the identity conversion.

There is no permitted conversion to the type boolean other than the identity conversion.

There is no permitted conversion from the type boolean other than the identity conversion and string conversion.

There is no permitted conversion other than string conversion from class type *S* to a different class type *T* if *S* is not a subclass of *T* and *T* is not a subclass of *S*.

There is no permitted conversion from class type *S* to interface type *K* if *S* is final and does not implement *K*.

There is no permitted conversion from class type *S* to any array type if *S* is not Object.

There is no permitted conversion other than string conversion from interface type *J* to class type *T* if *T* is final and does not implement *J*.

There is no permitted conversion from interface type *J* to interface type *K* if *J* and *K* declare methods with the same name and the same signature but different return types.

There is no permitted conversion from any array type to any class type other than Object.

There is no permitted conversion from any array type to any interface type.

There is no permitted conversion from array type *SC*[] to array type *TC*[] if there is no permitted conversion other than a string conversion from *SC* to *TC*.

5.2 Assignment Conversion

Assignment conversion occurs when the value of an expression is assigned to a variable: the type of the expression must be converted to the type of the variable. Assignment contexts allow the use of an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), or a widening reference conversion (§5.1.4). In addition, a narrowing primitive conversion may be used if all of the following conditions are satisfied:

- The expression is a constant expression of type int.
- The type of the variable is byte, short, or char.

- The value of the expression (which is known at compile time, because it is a constant expression) is representable in the type of the variable.

If the type of the expression cannot be converted to the type of the variable by a conversion permitted in an assignment context, then a compile-time error occurs.

If the type of an expression can be converted to the type a variable by assignment conversion, we say the expression (or its value) is *assignable* to the variable or, equivalently, that the type of the expression is *assignment compatible* with the type of the variable.

An assignment conversion never causes an exception. (Note, however, that an assignment may result in an exception in a special case involving array elements —see §10.9 and the last example in §5.3.2.)

The compile-time narrowing of constants means that code such as:

```
byte theAnswer = 42;
```

is allowed. Without the narrowing, the fact that the integer literal 42 has type int would mean that a cast to byte would be required:

```
byte theAnswer = (byte)42;          // Cast is permitted but not required
```

A value of primitive type must not be assigned to a variable of reference type; an attempt to do so will result in a compile-time error. A value of type boolean can be assigned only to a variable of type boolean.

The following test program contains examples of assignment conversion of primitive values:

```
class Test {
    public static void main(String args[]) {
        short s = 12;                   // narrow 12 to short
        float f = s;                    // widen short to float
        System.out.println("f=" + f);
        char c = '\u0123';
        long l = c;                     // widen char to long
        System.out.println("l=0x" + Long.toString(l,16));
        f = 1.23f;
        double d = f;                   // widen float to double
        System.out.println("d=" + d);
    }
}
```

and produces the following output:

```
f=12            // should be 12.0 ???
i=0x123
d=1.23457
```

The following test, however, produces compile-time errors:

```
class Test {
    public static void main(String args[]) {
        short s = 123;
        char c = s;          // error: would require cast
        s = c;               // error: would require cast
    }
}
```

because not all short values are char values, and neither are all char values short values.

A value of reference type must not be assigned to a variable of primitive type; an attempt to do so will result in a compile-time error.

A null (§14.7.2) may be assigned to any reference type, resulting in a null reference of that type.

Here is a sample program illustrating assignments of references:

```
public class Point { int x, y; }

public class Point3D extends Point { int z; }

public interface Colorable {
    void setColor(int color);
}
public class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String args[]) {
        // assignments to variables of class type
        Point p = new Point();

        p = new Point3D();              // ok: because Point3d is a
                                        // subclass of Point

        Point3D p3d = p;                // error: will require a cast
                                        // because a Point might not
                                        // be a Point3D (even though it
                                        // is dynamically in this
                                        // example.)
        // assignments to variables of type Object
        Object o = p;                   // ok: any object to Object
        int[] a = new int[3];
        Object o2 = a;                  // ok: an array to Object
        // assignments to variables of interface type
        ColoredPoint cp = new ColoredPoint();
        Colorable c = cp;               // ok: ColoredPoint implements
                                        // Colorable
        // assignments to variables of array type
        byte[] b = new byte[4];
        a = b;                          // error: these are not arrays
                                        // of the same primitive type
        Point3D[] p3da = new Point3D[3];
        Point[] pa = p3da;              // ok: since we can assign a
                                        // Point3D to a Point
        p3da = pa;                      // error: (cast needed) since
                                        // a Point can't be assigned
                                        // to a Point3D
    }
}
```

Assignment of a value of compile-time reference type *S* (source) to a variable of compile-time reference type *T* (target) is checked as follows:

- If *S* is a class type:
  - If *T* is a class type, then *S* must be the same class as *T*, or *S* must be a subclass of *T*, or a compile-time error occurs.
  - If *T* is an interface type, then *S* must implement interface *T*, or a compile-time error occurs.
  - If *T* is an array type, then a compile-time error occurs.
- If *S* is an interface type:
  - If *T* is a class type, then *T* must be Object , or a compile-time error occurs.
  - If *T* is an interface type, then *T* must be the same interface as *S*, or *T* a superinterface of *S*, or a compile-time error occurs.
  - If *T* is an array type, then a compile-time error occurs.
- If *S* is an array type *SC*[], that is, an array of components of type *SC*:
  - If *T* is a class type, then *T* must be Object, or a compile-time error occurs.
  - If *T* is an interface type, then a compile-time error occurs.
  - If *T* is an array type *TC*[], that is, an array of components of type *TC*, then a comoiler-time error occurs unless one of the following is true:
    - *TC* and *SC* are the same primitive type.
    - *TC* and *SC* are both reference types and type *SC* is assignable to *TC*.

See §8 for the detailed specifications for classes, §9 for interfaces, and §10 for arrays.

The following test program illustrates assignment conversions on reference values, but fails to compile because it violates the preceding rules, as described in its comments. This example should be compared to the preceding one.

```
public class Point { int x, y; }

public interface Colorable { void setColor(int color); }

public class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String args[]) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();

    // ok because ColoredPoint is a subclass of Point:
        p = cp;
    // ok because ColoredPoint implements Colorable:
        Colorable c = cp;
    // the following cause compile-time errors because
    // we cannot be sure they will succeed, depending on the
    // run-time type of p; a run-time check will be
    // necessary for a conversion and must be indicated
    // by including a cast:
        cp = p;         // p might be neither a ColoredPoint
                        // nor a subclass of ColoredPoint
        c = p;          // p might not implement Colorable
    }
}
```

Here is another example involving assignment of array objects:

```
class Point { int x, y; }

class ColoredPoint extends Point { int color; }

class Test {
    public static void main(String args[]) {
        long[] veclong = new long[100];
        Object o = veclong;                     // ok
        Long l = veclong;                       // compile-time error
        short[] vecshort = veclong;             // compile-time error
        Point[] pvec = new Point[100];
        ColoredPoint[] cpvec = new ColoredPoint[100];
        pvec = cpvec;                           // ok
        pvec[0] = new Point();                  // would cause exception
        cpvec = pvec;                           // compile-time error
    }
}
```

In this example:

- You can't assign veclong to a Long variable, because Long is a class type (§19.7). You can assign arrays only to variables with compatible array types, or to a variable of type Object.
- You can't assign veclong to vecshort, because they are arrays of primitive type, and short and long are not the same primitive type.
- You can assign cpvec to pvec, because you can assign a reference value of type ColoredPoint to a Point. The subsequent assignment of the new Point to pvec would throw an ArrayStoreException (if the program were fixed to compile), because a ColoredPoint array can't have a Point as the value of a component.
- You can't assign pvec to cpvec, because you can't assign a reference value of type Point to a ColoredPoint. If the run-time type of pvec was Point[], and the assignment was allowed, a simple reference to a component of cpvec, for example, cpvec[0], could return a Point, and a Point is not a ColoredPoint, violating the type system. You must first ensure with a cast (§5.4, §14.15) that pvec references a ColoredPoint[]. Seethe examples in §5.5.2.

5.3 Method Invocation Conversion

Method invocation conversion is applied to each argument value in a method or constructor invocation (§14.11): the type of the argument expression must be converted to the type of the corresponding parameter. Method invocation contexts allow the use of an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), or a widening reference conversion (§5.1.4).

Method invocation conversions specifically do not the include implicit narrowing of integer constants which is part of assignment conversion (§5.2). The Java designers felt that including these implicit narrowing conversions would add additional complexity to the overloaded method matching resolution process (§14.11.2). Thus the example:

```
class Test {
    static int m(byte a, int b) { return a+b; }
    static int m(short a, short b) { return a-b; }
    public static void main(String args[]) {
        System.out.println(m(12, 2));                    // compile-time error
    }
}
```

causes a compile-time error because the integer literals 12 and 2 have type int, so neither method m matches under the rules of (§14.11.2). A language that included implicit narrowing of integer constants would need additional rules to resolve cases like this example.

5.4 Casting Conversions

Casting conversion is applied to the operand of a cast operator (§14.15): the type of the operand expression must be converted to the type explicitly named by the cast operator. Casting contexts allow the use of an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), a narrowing primitive conversion (§5.1.3), a widening reference conversion (§5.1.4), or a narrowing reference conversion (§5.1.5). Thus casting conversions are more inclusive than assignment or method invocation conversions: a cast can do any permitted conversion other than a string conversion.

A value of reference type cannot be cast to a primitive type. Some casts can be proven incorrect at compile time; such casts result in a compile-time error. The detailed rules for compile-time correctness checking of a casting conversion of a value of compile-time type *S* (source) to a compile-time type *T* (target) are as follows:

- If *S* is a class type:
  - If *T* is a class type, then *S* and *T* must be related classes, that is, *S* and *T* must be the same class , or *S* a subclass of *T,* or *T* a subclass of *S*; otherwise a compile-time error occurs.
  - If *T* is an interface type:
    - If *S* is not a final class (§8.1.3), then the cast is always correct at compile time (because even if *S* doesn't implement *T*, a subclass of *S* might).
    - If *S* is a final class (§8.1.3), then *S* must implement *T*, or a compile-time error occurs.
  - If *T* is an array type, then *S* must be the class Object, or a compile-time error occurs.
- If *S* is an interface type:
  - If *T* is a class type that is not final (§8.1.3), then the cast is always correct at compile time (because even if *T* doesn't implement *S*, a subclass of *T* might).
  - If *T* is a class type that is final (§8.1.3), then *T* must implement *S*, or a compile-time error occurs.
  - If *T* is an interface type and if *T* and *S* contain methods with the same signature (§8.4.3) but different return types, then a compile-time error occurs.
- If *S* is an array type *SC*[], that is, an array of components of type *SC*:
  - Tf *T* is a class type, then if *T* is not Object, then a compile-time error occurs (because Object is the only class type to which arrays can be assigned).

- If *T* is an interface type, then a compile-time error occurs (because arrays do not implement any interfaces).
- If *T* is an array type *TC*[], that is, an array of components of type *TC*, then a compile-time error occurs unless one of the following is true:
  - *TC* and *SC* are the same primitive type.
  - *TC* and *SC* are reference types and type *SC* can be cast to *TC*, by these rules.

See §8 for the detailed specifications of classes, §9 for interfaces, and §10 for arrays.

If a cast to a reference type is not a compile-time error, there are two cases:

- The cast can be determined to be correct at compile time. A cast from compile-time type *S* to compile-time type *T* is correct at compile time if and only if *S* can be converted to *T* by assignment conversion (§5.2).
- The cast requires a run-time validity check. If the value at run time is null, then the cast is allowed. Otherwise, let *R* be the class of the object referred to by the run-time reference value, and let *T* be the type named in the cast operator. A cast conversion must check, at run time, that the class *R* is assignment compatible with the type *T*, using the algorithm specified in §5.2, using the class *R* instead of the compile-time type *S* as specified there, namely:
  - If *R* is an ordinary (non-array) class, then
    - If *T* is a class type, then *R* must be the same class (§4.4) as *T*, or a subclass of *T*, or a run-time exception is thrown.
    - If *T* is an interface type, then *R* must implement (§8.1.5) interface *T*, or a run-time exception is thrown.
    - If *T* is an array type, then a run-time exception is thrown.
  - *R* cannot be an interface, because there are no instances of interfaces, only of classes and arrays (§4.3.1).
  - If *R* is a class representing an array type *RC*[], that is, an array of components of type *RC*, then
    - iI *T* is a class type, then if *T* is not Object (§4.3.2, §19.1), a run-time exception is thrown.
    - If *T* is an interface type, then a run-time exception is thrown (because no array implements any interface—this case could slip past the compile-time checking if, for example, a reference to an array were stored in a variable

j.12345.doc 64 Wed Mar 27 08:55:12 1996

of type Object).

- If *T* is an array type *TC*[], that is, an array of components of type *TC*, then a run-time exception is thrown unless one of the following is true:
  - *TC* and *RC* are the same primitive type.
  - *TC* and *RC* are reference types and type *RC* can be cast to *TC*, by the *compile-time* rules for casting (not these run-time rules).

If a run-time exception is thrown, it is a ClassCastException (§11, §19.21).

Here are some examples of casting conversions of reference types, similar to the example in §5.2:

```
public class Point { int x, y; }

public interface Colorable { void setColor(int color); }

public class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

final class EndPoint extends Point { }

class Test {
    public static void main(String args[]) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        Colorable c;
    // the following may cause errors at run time because
    // we cannot be sure they will succeed; this possibility
    // is suggested by the casts:
        cp = (ColoredPoint)p;           // p might not reference an
                                        // object which is a ColoredPoint
                                        // or a subclass of ColoredPoint
        c = (Colorable)p;               // p might not be Colorable
    // the following are incorrect at compile time because
    // they can never succeed as explained in the text.
        Long l = (Long)p;               // compile-time error #1
        EndPoint e = new EndPoint();
        c = (Colorable)e;               // compile-time error #2
    }
}
```

Here the first compile-time error occurs because the class types Long and Point are unrelated (that is, they are not the same, and neither is a subclass of the other), so a cast between them will always fail.

The second compile-time error occurs because a variable of type EndPoint can never reference a value that implements the interface Colorable. This is because EndPoint is a final type, and a variable of a final type always holds a value of the same run-time type as its compile-time type. Therefore the run-time type of variable e must be exactly the type EndPoint, and type EndPoint does not implement Colorable.

Here is an example involving arrays:

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return "(" + x + "," + y + ")"; }
}

public interface Colorable { void setColor(int color); }

public class ColoredPoint extends Point implements Colorable {
    int color;
    ColoredPoint(int x, int y, int color) {
        super(x, y); setColor(color);
    }
    public void setColor(int color) { this.color = color; }
    public String toString() {
        return super.toString() + "@" + color;
    }
}

class Test {
    public static void main(String args[]) {
        Point[] pa = new ColoredPoint[4];
        pa[0] = new ColoredPoint(2, 2, 12);
        pa[1] = new ColoredPoint(4, 5, 24);
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.print("cpa: {");
        for (int i = 0; i < cpa.length; i++)
            System.out.print(" " + cpa[i] + ",");
        System.out.println(" }");
    }
}
```

This example compiles without errors and produces the output:

```
cpa: { (2,2)@12, (4,5)@24, null, null, }
```

The following example uses casts to compile, but it throws exceptions at run time, because the types are incompatible.

```
public class Point { int x, y; }

public interface Colorable { void setColor(int color); }

public class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) {
        this.color = color;
```

```
	}
}

class Test {
	public static void main(String args[]) {
		Point[] pa = new Point[100];
		// following will throw a ClassCastException:
		ColoredPoint[] cpa = (ColoredPoint[])pa;
		System.out.println(cpa[0]);

		int[] shortvec = new int[2];
		Object o = shortvec;

		// following will cause a run-time error:
		Colorable c = (Colorable)o;
		c.setColor(0);
	}
}
```

5.5 String Conversion

String conversion applies only to the operands of the binary + operator when one of the arguments is a String. In this single special case, the other argument to the + is converted to a String, and a new String which is the concatenation of the two strings is the result of the +. String conversion is described in more detail in the description of the string concatenation + operator (§14.17.1).

5.6 Numeric Promotions

Numeric promotion is applied to the operands of an arithmetic operator. Numeric promotion contexts allow the use of an identity conversion (§5.1.1) or a widening primitive conversion (§5.1.2).
Numeric promotions are used to convert the operands of a numeric operator to a common type so that an operation can be performed. The two kinds of *numeric promotion* are unary numeric promotion and binary numeric promotion. The analogous conversions in C are called "the usual unary conversions" and "the usual binary conversions."

Numeric promotion is not a general feature of Java, but rather a property of the specific definitions of the built-in operations.

5.6.1 Unary Numeric Promotion

Some operators apply *unary numeric promotion* to a single operand, which must denote a value or a variable of a numeric type:

- If the operand is of compile-time type byte, short, or char, unary numeric promotion promotes it to a value of type int by a widening conversion (§5.1.2).
- Otherwise a unary numeric operand remains as is and is not converted.

Unary numeric promotion is performed on:

- The index expression in array access expressions (§14.12)
- The dimension expression in array creations (§14.9);
- Operands of the unary plus + (§14.14.3) and unary minus - (§14.14.4) operators
- The operand of the bitwise complement operator ~ (§14.14.5); and
- Each operand, separately, of the shift operators >>, >>>, and << (§14.18), so that a long shift distance (right operand) does not promote the value being shifted to long.

Here is a test program that includes examples of unary numeric promotion:

```
class Test {
    public static void main(String args[]) {
        byte b = 2;
        int a[] = new int[b];      // dimension expr. promotion
        char c = '\u0001';
        a[c] = 1;                  // index expression promotion
        a[0] = -c;                 // unary - promotion
        System.out.println("a: " + a[0] + "," + a[1]);

        b = -1;
        int i = ~b;                // bitwise complement promotion
        System.out.println("~0x" + hex(b) + "==0x" + hex(i));

        i = (int)/*bug*/(b << 4L);
        System.out.println("0x" + hex(b) + "<<4L==0x" + hex(i));
    }
    // bug: use new String.toHexString method instead
    static String hex(int val) {
        String s = (val &~ 0xf) != 0 ? hex(val>>>4) : "";
        val &= 0xf;
        return s + (char)((val<10 ? '0':'a'-10)+val);
    }
}
```

This test program produces the output:

```
a: -1,1
~0xffffffff==0x0
0xffffffff<<4L==0xfffffff0
```

5.6.2 Binary Numeric Promotion

When an operator applies *binary numeric promotion* to a pair of operands, each of which must denote a value or a variable of a numeric type, the following rules apply, in order, using widening conversion (§5.1.2) to convert operands as necessary:

- If either operand is of type double, the other is converted to double.
- Otherwise, if either operand is of type float, the other is converted to float.
- Otherwise, if either operand is of type long, the other is converted to long.
- Otherwise, both operands are converted to type int.

Binary numeric promotion is performed on the operands of:

- The multiplicative operators *, / and % (§14.16)
- The addition and subtraction operators for numeric types + and - (§14.17.2)
- The numerical comparison operators <, <=, >, and >= (§14.19.1)
- The numerical equality operators == and != (§14.20.1)
- The integer bitwise operators &, ^ and | (§14.21.1)
- In certain cases, the conditional operator ? : (§14.24)

An example of binary numeric promotion was shown earlier in §5.1. Here is another:

```
class Test {
    public static void main(String args[]) {
        int i = 0;
        float f = 1;
        double d = 2;

        // i*f promoted to float*float, then
        // float==double promoted to double==double
        if (i * f == d)
            System.out.println("oops");
        // char&byte promoted to int&int
        byte b = 0x1f;
        char c = 'a';
        int control = 'a' & b;
        System.out.println(hex(control));

        // int:float promoted to float:float
        f = (b==0) ? f : 37;
        System.out.println(f);
    }
    static String hex(long i) {
        return Long.toString(i, 16);
    }
}
```

which produces the output:

```
1
37
```

The example converts the ASCII character a to the ASCII control-a, by masking off all but the low 5 bits of the character. The 1 is the numeric value of this control character.